Aspect Security

Ravishekhar Gopalan
College of Computer Science
Northeastern University
Boston

Prof. Karl Lieberherr
College of Computer Science
Northeastern University
Boston

1. Abstract

This paper deals with security of Aspects weaved into code. As AOP becomes a mainstream programming approach to software development, a controlled and safe approach is required for efficient development and design using aspects. Since aspects by their very definition imply compartmentalization, weaving together aspects written for different concerns by different people can be error-prone and might create the same sort of problems which AOP aims to solve. Especially with regard to security, the interweaving of aspects might unknowingly open a backdoor into the system. Determination of such vulnerability can be extremely difficult. Hence we need a stronger safety net than those required by non-AOP languages which require certain practices to be followed along the aspect-oriented development cycle. This paper attempts to define such practices which lead to the development of secure aspects and approaches to achieving the same. It also outlines a proof of concept for a general purpose approach to securing aspect code.

1. Introduction

Security is a non-functional lateral requirement of any system. Even though there are daily reports of attacks which exploiting vulnerabilities in code, security is not "built-in" to the system from the design stage when it would have been most effective, but applied or rather "patched" onto the system at the end of the SDLC[2]. This has led to a preference of non-invasive techniques for securing code. This has further widened the gap between the development and security cycle. While non-invasive security principles are quite effective, they cannot nullify an existing vulnerability in the system. In order to secure a system, security has to be built-in from the beginning [3]. Security can be applied to a system in different ways and an understanding of these will aid in applying the right kind of security in the right context. The first level of security is provided by the language and the runtime which is being used to build the software. Next come Business rules which fall into the category of Domain Security and the third approach is non-invasive security application using security policies external to the software.

The focus of this paper is the security features provided by the language. Specifically, in the light of AOP, we present an approach of building-in security rules into the language itself using the abc compiler. A point to be noted is that our analysis of AOP will be limited to OO systems. This is basically because AOP has more support in the JAVA community than in any other programming language community.

The rest of the paper is organized as follows. Section 2 elaborates the security of an aspect and the methods to achieve it. Section 3 gives a brief overview of the abc compiler from Oxford University, UK and finally the actual implementation is discussed in Section 4. Future work and References are given in Sections 5 and 6 respectively.

2. Aspect Security

Ideally, the programming language should provide necessary security related constructs which can be used to tag the behavior of any subroutine. The security vulnerabilities can be reduced in part if every method declares its behavior, thus helping the compiler. Alternatively, metadata could be used to tag the type of behavior of each method.

All these approaches indirectly pass on part of the burden of security checking onto the language runtime. In effect, the target is an intelligent compiler which is able to statically determine any current or possible future breach of security. Wherever the compiler is unable to determine statically whether a piece of code can be exploitable in future for a particular flaw, it should automatically weave in code which performs checks at runtime. This might be an ambitious project because the field of compilers is quite old and research has been going on for some decades. But AOP compilers are a recent development and it would be advisable to build in security checks at code weaving time.

That brings up the next question; "*What is a secure aspect?*"
Before the security of an aspect is defined, it is important to note that securing software usually involves imposing restrictions on either the operation of the software or on the usage of the software. Extending this definition of security to aspects would translate to restrictions being imposed on aspects which would take away part of the power of AOP.

While this may be true in some cases, what exactly is being proposed is that the usage of AOP in a software system should not overlap with the usage of OOP. An example is the creation of an inheritance hierarchy of classes. This is possible using both AOP and OOP. In AOP, this can be done using the "*declare parents*" construct whereas this is supported in OOP using constructs like "extends" or ":" depending on the language. Using AOP and OOP to achieve the same would inevitably lead to confusion. It would be better to play to the strengths of both by keeping them independent.

In this light, a secure aspect can be defined as an aspect which

- Does not modify the data of the object on which it is operating on
- Does not violate the OO design of the system which it is operating on
- Does not modify the behavior of the object on which it is operating on

The popularity of OO grew because it allowed data segregation and ownership. It would be a folly if we introduce AOP to modify the same data without the knowledge of the object when it interacts with other components of the system. Hence, this restriction on an aspect to modify an object's data.

OO design violation would involve modification of the type hierarchy of objects. Modification of behavior would mean adding to the existing of behavior of objects. Both of these can be achieved by OO techniques of inheritance. For example,

modification of behavior of an object can be achieved by creating a sub-class in which the method of the parent class is overridden. This does not require a new AO paradigm to achieve.

But does this mean that an aspect should not do anything? Not at all. The power of aspects does not lie in providing the same functionality as an object. But rather, it lies in separating the concerns which are tied to that object. In effect an aspect is extending the functionality of the object in another plane, another dimension. And a secure object is defined as an aspect which does just this without overlapping on the OO part of the system.

Basically, a secure aspect should add to the behavior at the join-point it is operating on i.e. perform check on the data validity or method arguments. But it should be an inspector and should not modify the data of the object. To be more precise, an aspect should add behavior at a join point and not to an object.

3. Achieving aspect security

In order to check the security of an aspect, there are a couple of approaches that can be adopted.

- § Write an aspect that influences other aspects
- § Modify the aspect compiler to check the aspect security.
- § Use meta-AOP

The first approach appears to be the easiest one. Declare an aspect with the highest precedence among all the aspects operating at that join point and make it check the remaining aspects. But the hurdle in this case would be to make all other aspects operating at a particular join point available to the "master" aspect which has the highest precedence.

The third approach specifies the usage of meta-AOP. This would be similar to meta-classes of OOP. Both the above two approaches would involve adding to the point cut language. For e.g. the language should have the capability to declare a point cut that would be able to select a call to all aspects at that join point. This is provided in AspectJ 1.2 with the advice execution join point, but this is not sufficient to achieve what we want.

The third approach being the simplest and the easiest is selected for this POC. This involves intervening during compile time running the aspect code through an aspect checker.

For reasons of simplicity and extensibility, the abc (Aspect Bench Compiler) compiler from Oxford University is the best choice for implementation. abc is built with extensibility in mind. Also it is conducive to the writing of an Aspect checker by providing the AST for the code and instead of the checker having to parse the aspect code.

4. abc Compiler

The Aspect Bench Compiler (*abc*) is an extensible compiler which supports the full AspectJ syntax. Its front-end is built using the Polyglot framework. Polyglot is an extensible

front-end for Java that performs all the semantic checks required by the language. It is structured as a list of passes that rewrite an AST, and build auxiliary structures such as a symbol table and type system. The backend is built using the Soot framework [4]. Soot is a Java byte code analysis toolkit based around the Jimple IR, a typed, three-address, stack-less code. Jimple is low-level enough for point cut matching, in that the granularity of any join point is at least one entire Jimple statement. Soot can produce Jimple from both byte code and Java source code. As output, Soot generates Java byte code.

The high level architecture of the abc compiler is shown below [1].

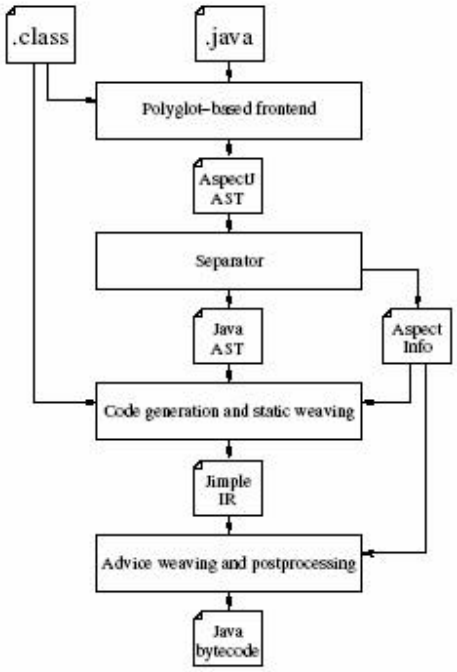

Fig. 1. Architecture of abc compiler.

The following is a proposed point of change to the abc compiler.

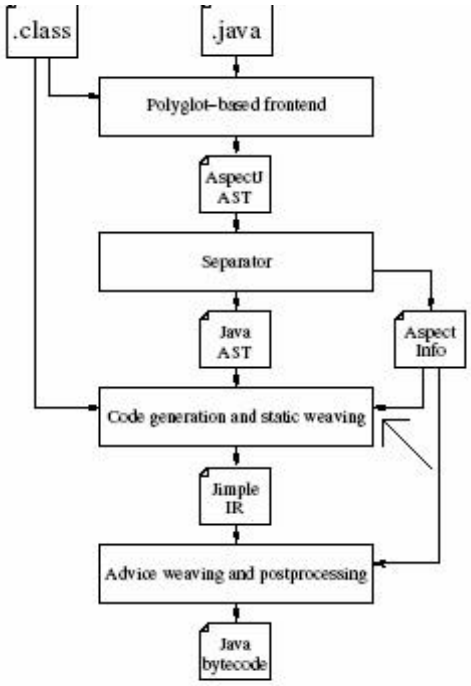

Fig. 2. Proposed change to abc.

5. Implementation

In order to modify the compiler to verify the aspects before weaving, all the aspects need to be run through an Aspect Checker. The implementation of an Aspect checker as shown in the proposed change in the previous section, is to introduce a call to the Aspect Checker in the compiler.

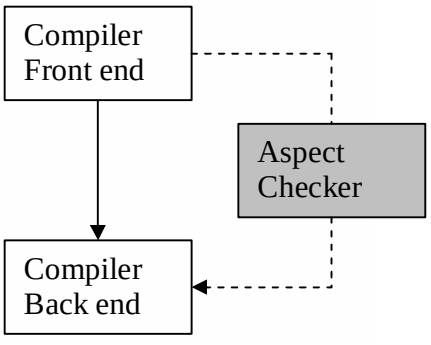

Fig. 3. Introduction of Aspect Checker.

The aspect checker should check for the security level of the aspect based on the conditions specified above. For this Proof of Concept, we are checking if the aspect is modifying the data of the object on which it is operating. If so, then compilation fails. The check is based on the assumption that all data members f the object are private and the data is modified only through public setter methods. It is easy to circumvent this check, but a stronger checking is left as future work and not included as part of this Proof of Concept.

Shown below is a code snippet which checks for the setter method call within the aspect. Each statement of every method of every aspect is checked and if a match is found, an exception is thrown.

```
if (
    ( method.getParameterCount() > 0 )  &&
    ( getTargetType(method) = =
            getTargetType(stmt) ) &&
    (
aSootMethod.getName().startsWith("set") )
  )
    throw new InsecureAspectException();
```

Fig. 4. Code Snippet

The design of the aspect checker is shown below. It takes into account that each class might have aspects which might check some behavior specific to that class. Hence these individual specific aspect checkers are introduced after the Global Aspect Checker checks the aspects for security.

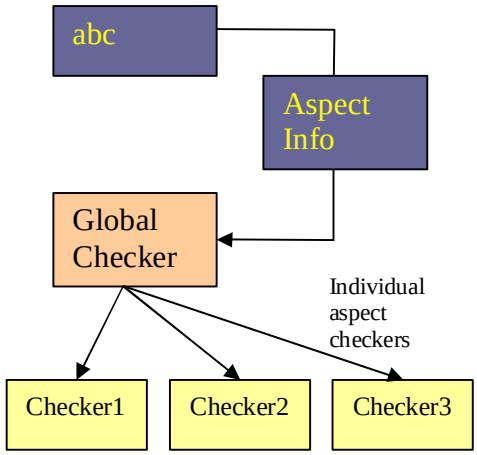

Fig.5. Design of Aspect Checker

The abc compiler passes in all the aspects to the Global Aspect Checker which performs checks common to all aspects. The Global aspect checker then passes on these aspects to the individual aspect checkers which are specific for a particular class.

6. Future Work

The Proof of Concept has been provided in this project. But it is by no means an exhaustive work as a more rigorous security net for the aspects still is pending. The following are some of the issues which need to be addressed in the future.

- § Incorporate DJ into the aspect checker
  Currently, the code manually traverses the AST. DAJ traversals might be incorporated to facilitate this task.
- § Perform more rigorous checking for the aspects.
  This PoC checks the aspects only for a particular rule about data

modification. All rules need to be incorporated in to the checker.

- § Detailed definition for aspect security
  A definition for a "secure" aspect has been provided in this paper. A more comprehensive definition is pending.
- § Giving warning instead of throwing exception
  A minor modification which can be done is to throw a warning instead of an exception so that the compilation goes through but at the same tome the user is informed about the security violation.
- § Making security as an input flag
  Security can be passed in as an input flag so that the aspect checker is invoked only if this flag is passed in.
- § Dynamic Weaving of Aspect Checking code.
  This project deals with static checking of aspects. In some cases, code has to be dynamically woven in so that precautionary measures can be taken about those vulnerabilities which require runtime information.

7. References

[1] **Building the abc AspectJ compiler with Polyglot and Soot**
abc Technical Report No. abc-2004-2
Pavel Avgustinov et al

[2] **SecureUML: A UML-Based Modeling Language for Model-Driven Security**
Torsten Lodderstedt, David Basin, and Jürgen Doser

[3] **Building Secure Software**
John Viega, Gary McGraw

[4] **abc : An extensible AspectJ compiler**
abc Technical Report No. abc-2004-1